ReactiveFnJ: A Choreographed Model for Fork-Join Workflow in Serverless Computing

Urmil Bharti (urmil.bharti@rajguru.du.ac.in)
Shaheed Rajguru College of Applied Sciences for Women https://orcid.org/0000-0003-2655-3001
Anita Goel
University of Delhi
S. C. Gupta
Indian Institute of Technology

Research Article

Keywords: Serverless computing, FaaS, Event-driven function composition, Choreography, Parallel computing, Distributed computing, Fork and Join, Orchestration

Posted Date: January 10th, 2022

DOI: https://doi.org/10.21203/rs.3.rs-942960/v1

Version of Record: A version of this preprint was published at Journal of Cloud Computing on April 24th, 2023. See the published version at https://doi.org/10.1186/s13677-023-00429-3.

Abstract

Function-as-a-Service (FaaS) is an event-based reactive programming model where functions run in ephemeral stateless containers for short duration. For building complex serverless applications, function composition is crucial to coordinate and synchronize the workflow of an application. Some serverless orchestration systems exist, but they are in their primitive state and do not provide inherent support for non-trivial workflows like, Fork-Join. To address this gap, we propose a fully serverless and scalable design model *ReactiveFnJ* for Fork-Join workflow. The intent of this work is to illustrate a design which is completely choreographed, reactive, asynchronous, and represents a dynamic composition model for serverless applications. Our design uses two innovative patterns, namely, Relay Composition and Master-Worker Composition to solve execution time-out challenges. As a Proof-of-Concept (PoC), the prototypical implementation of Split-Sort-Merge use case, based on Fork-Join workflow is discussed and evaluated. The *ReactiveFnJ* handles embarrassingly parallel computations, and its design does not depend on any external orchestration services, messaging services, and queue services. *ReactiveFnJ* facilitates in designing fully automated pipelines for distributed data processing systems, satisfying the Serverless Trilemma in true essence. A file of any size can be processed using our effective and extensible design without facing execution time-out challenges. The proposed model is generic and can be applied to a wide range of serverless applications that are based on the Fork-Join workflow pattern. It fosters the choreographed serverless composition for complex workflows. The proposed design model is useful for software engineers and developers in industry and commercial organizations, total solution vendors and academic researchers.

1. Introduction

Serverless computing is an emerging paradigm and is gaining popularity in the cloud owing to its simplicity, billing model, and inherent elasticity. This cloud computing execution model greatly simplifies the usage of cloud resources and suits well to highly scalable, event-driven applications in the cloud. Serverless architecture is especially effective at supporting modern applications with unpredictable scale and user demand [2].

The FaaS programming model of serverless allows programmers to develop cloud applications as individual functions that can run and scale independently. This model is event-driven since functions are activated in reaction to specific cloud events like, a state change in an object store, receipt of a message, a file upload, or insertion of a record in database. Though FaaS looks like a promising option for deploying cloud applications, it has few limitations also. Most notably, FaaS functions are stateless, short-lived, and cannot communicate directly with each other [15] [24]. Thus, executing complex, burst-parallel jobs pose a significant challenge for serverless execution frameworks [13]. The composition of workflows in such jobs require extensive fine-grained communication and synchronization between independent functions that is challenging to implement in a serverless framework [8]. These FaaS challenges force developers to resort to alternate ways to establish function communication like publish-

subscribe, passing data over some slow and expensive storage medium, or serverless orchestration services, but all these alternatives yield too high latency and cost [27].

At a high level, there are two approaches for function composition in a serverless application: orchestration and choreography [28], as shown in Fig. 1. In orchestration, a controller module orchestrates and controls the interaction between serverless functions. The controller module governs the flow according to the needs of the business logic. The choreography model is an event-driven paradigm in which every function works autonomously as a loosely coupled service. The functions work in a pipeline based on the triggered events. Each function performs its task, and its completion triggers the next function/s down the pipeline. In an event-driven architecture where each component plays a more architecturally aware role, the choreography model is used in the design of workflows instead of an orchestration model [4]. [23].

The Fork-Join model is a programming method that exploits parallelism in applications based on inherent *divide and conquer* algorithms [22]. This execution model has already been successfully used for building parallel systems where an incoming task splits into subtasks that are processed by a set of parallel servers. The implementation of the Fork-Join model becomes more practical in serverless computing as these platforms are inherently scalable and do not need resource provisioning in advance. Since horizontal scaling in serverless is entirely automatic, elastic, and managed by its provider, dynamic parallel processing in Fork-Join can best exploit these characteristics.

Currently, Fork-Join workflow cannot be composed using any of the available serverless orchestration services [4] like Amazon Step Functions (ASF) (December 2016), Azure Durable Functions (June 2017), and IBM Composer (October 2017). These services lack the ability to dynamically launch functions in parallel. Though ASF, the most mature and performant project [13], supports the *Parallel* state type to execute tasks in parallel. However, the application developers must list all the tasks to execute in parallel in an array in the state machine, thereby restricting the flexibility of the concurrency level [34].

In our research, we present *ReactiveFnJ*, a fully choreographed, vendor-neutral, and platform-independent serverless design model for Fork-Join workflow. The *ReactiveFnJ* uses an innovative algorithmic design that is purely event-driven, reactive, ST-Safe, and is free from hard execution time limits imposed by a serverless provider. The *ReactiveFnJ* handles embarrassingly parallel computations, but it does not depend on any external orchestration services, messaging services, and queue services. This model is generic, and several other use cases can be implemented by substituting specialized FaaS functions in our PoC implementation. *ReactiveFnJ* model is the first work that exhibits all the characteristics mentioned above to the very best of our knowledge. Our design and its implementation will also lead us to find answers of the following research questions:

RQ1: Does any serverless design exist for long running computation?

RQ2: Does the Serverless challenge of executing a complex, burst-parallel workflow can be addressed programmatically without using any external orchestration service?

RQ3: Is it possible to design a serverless system using pure event-driven architecture for complex workflows?

This manuscript is organized into several sections. Section 2 presents the technical background, motivation behind the proposed *ReactiveFnJ* serverless design model and summarizes our contributions. The detailed design and algorithms related to our serverless Fork-Process-Join pipeline are described in Sect. 3. Section 4 gives a detailed description of the implementation of our design concepts in the AWS Lambda. Section 5 shows the evaluation insights of the proposed models, followed by a discussion and lessons in Sect. 6. The related works about the serverless applications implemented for parallel processing are described in Sect. 7. The concluding remarks and scope for future works are delineated in Sect. 8.

2. Background, Motivation, And Contribution

This section discusses current serverless challenges, the motivation behind this work, and the significant contributions of this research.

A. Serverless and its Challenges

The serverless model was originally designed to execute event-driven, stateless functions in response to user actions or changes in the storage tier, e.g., uploading a photo to Amazon S3, inserting/updating a record in DynamoDB, etc. [21]. Some recent works have shown that the large-scale parallelism and auto-scaling features provided by serverless platforms make them well-suited for burst-parallel fine-grained tasks and parallel computation workflows [12]. In essence, the FaaS model is apt for embarrassingly parallel computing use cases such as linear algebra [31], optimization algorithms [10], data analytics [30], and real-time machine learning classifications [9].

Building a complex serverless application with numerous short-lived, concurrent functions requires new design guidelines. Beyond simple examples, serverless applications need to be designed as a composition of functions. In most of the cases, a serverless workflow composition needs an orchestration service that provides a coordination mechanism between FaaS functions [16]. These coordination services automatically trigger the execution of each function in the workflow and synchronize their behaviors and states. FaaS orchestration services such as AWS Step Functions or IBM Composer offer limited capabilities to coordinate serverless functions [5]. For instance, even in the AWS Step function, there is no provision for multiple functions to synchronize in parallel when the number of parallel instances is dynamic. In place of orchestration services, developers use some indirect ways to synchronize the dynamic parallel execution of functions via notification services, queue services, and in-memory data store/cache services [12] but they have their own limitations and downsides.

Few researchers have come up with solutions for handling embarrassingly parallel computations in serverless. PyWren [19] uses its own ad-hoc external orchestration service, and ExCamera [11] relies on an external server to synchronize the parallel executions. These solutions add significant latency and cost.

Nevertheless, all the systems implemented so far for parallel computing do not comply with the four requirements claimed by Amazon for a serverless application: (i) No server management, (ii) Flexible scaling, (iii) Pay for value, and (iv) Automated high availability [4].

Barcelona-Pons. *et al*. argued that serverless functions follow a trigger-based model, so a FaaS composition should also be trigger-based [4]. This means that the termination of one or many functions should trigger the next stage (function) using asynchronous events in a workflow. From this perspective, any serverless composition achieving dynamic parallelism should also be trigger-based. Therefore, we emphasize that it is of utmost desire to build a serverless application as a complete reactive system of FaaS function compositions.

Any function composition is referred as Serverless Trilemma (ST) safe if it satisfies three main principles of Serverless Trilemma stated by Baldini et al. [3]: (1) Substitution - Each composition should behave like a function and could be substituted in any other pipeline, (2) Black-box - Each component of the workflow should be a black-box and abstracts from rest of the system i.e. implementation details of functions remain hidden from others, and (3) No Double billing - FaaS is a pay-per-use model, i.e., fine-grained resource measurement based on usage and there should not be double-billing of cloud function.

B. Fork-Join Model in Serverless

The generic Fork-Join model of parallel processing splits a compute-intensive task into smaller sub-tasks to process them parallelly using the available CPU cores [29]. Therefore, the Fork-Join design model renders execution speed-up by running forked tasks in parallel and combining their results. This model can be best utilized in serverless implementation for use cases like, sorting [20], searching [17], matrix multiplication [6], string matching, MapReduce patterns like counting and summing, Collating, and Filtering [14]. In all the mentioned cases where the volume of data fluctuates, and resource requirements cannot be anticipated, the pay-per-use model of serverless is best used.

The above use cases can be best implemented using the generic Fork-Join workflow of parallel processing in serverless but designing this scalable workflow efficiently is a challenge in serverless frameworks. Currently, function composition for Fork/Join is not directly supported by the existing serverless orchestration services [4]. The available orchestration services are not designed for managing parallel Fork-Join workflows in a scalable and efficient way.

C. Motivation

Execution time limit is a major constraint of FaaS that hinders its implementation to those applications where running time might go beyond the set time limits. We experimented in the AWS Lambda, a serverless computing service provided by Amazon Web Services and found that a *Single FaaS* function can only sort a file of size 560 kb in a default environment setting. If this function gets an input file of size more than 560 kb, it ceases to complete due to execution time out.

It is just an example, but there are many compute-intensive scientific and business applications where constituent functions may time-out before their completion and hence applications are unable to harness the power of serverless computing. There is a class of inherently parallel applications in any domain, where the initial task can be split into a large number of independent sub-tasks (Fork), and then each sub-task can be implemented as autonomous functions in serverless. This design brings up two challenges in serverless: (i) to provide some coordination mechanism to run in parallel a multitude of functions derived from a single task and (ii) to devise a synchronization mechanism to join the results of all split tasks and to prepare the aggregated output (Join).

One can attempt to use an existing orchestration service, but it may not be a viable solution as they do not support the execution of tasks in parallel when number of tasks are dynamically determined at runtime. As an example, Amazon Step Functions support the *Parallel* state type to execute tasks in parallel, but application developers have to provide all the tasks to execute in parallel in an array construct of the state machine [8]. This restricts the flexibility of concurrency level and hence its usage in scenarios, where launching of functions in parallel is dynamic in nature.

The main downsides of currently available orchestration services are (i) Billing is based on the number of transitions happening during workflow execution (ii) Latency issues when working complex workflows (iii) Non adherence to serverless trilemma (violation of substitution and double billing principles) (iv) extra efforts are required to build a workflow in the orchestration services.

To handle these problems, we aim to design and implement a function composition mechanism for Fork-Join workflow which can be used into a broad spectrum of applications.

D. Contributions

The main contribution of this work is an algorithm-based design for serverless Fork-Join workflow. We have proposed a trigger-based serverless design model namely, *ReactiveFnJ*, for Fork-Join workflow. This design model can be utilized in compute intensive and burst-parallel applications. Our proposed model employs innovative design patterns that can process a file of any size without being time-out. In *ReactiveFnJ*, multiple component synchronization and coordination is crafted by *Relay Composition* and *Master-Worker Composition* design patterns thereby making it a choreographed and pure event-driven system. *ReactiveFnJ* is an asynchronous dynamic serverless composition design model that fully exploits the scalability, availability, and built-in fault tolerance of serverless infrastructure.

To prove the feasibility and viability of *ReactiveFnJ* design, we build a prototypical implementation to sort a large input file as a proof of concept. We call this sorting requirement as Split-Sort-Merge (SSM) use case throughout this research article. In SSM, the main aim is to sort a data file of any size (theoretically) where individual records are of variable length. There are several approaches to sort a large data file but the main challenge here is to develop an approach for a serverless architecture, where serverless functions are stateless and have a constrained execution environment. There exist multiple traditional serverful deployment frameworks for SSM, like, Map-Reduce and Apache Spark however, these

frameworks suffer from cluster management, load balancing and task fairness issues [1]. Thus, developing/migrating these applications to serverless platforms illustrates unique opportunities.

After the successful implementation of SSM, it can be claimed that applications designed using *ReactiveFnJ* will not have dependency on any external orchestration service, will not time out and will be free from vendor lock-in problems.

To summarize, the present work makes the following notable contributions:

- A pure event-driven choreographed system design for Fork-Join workloads suitable for serverless deployments.
- State of the art serverless function composition model satisfying all competing constraints of ST Trilemma (ST-Safe).
- Employs asynchronous push-based design exploiting recursive calling of stateless functions to handle long running compute intensive workloads.
- Self-driven function composition mechanism without relying on any external orchestration service to handle the complexities of state management.
- An Algorithm-level design solution that overcomes the execution time limitation imposed by serverless providers.

Our approach shall be a tipping point where a single machine/container is not big enough to perform a big computation and a serverless function fails for the same reason. Using our proposed novel algorithmic design approach, it shall be feasible to execute burst-parallel, compute intensive applications having large data-at-scale by leveraging the scalability benefit of FaaS.

3. Reactivefnj: Design

This section presents the design of *ReactiveFnJ* in detail. The main design components, their interaction, design challenges, and employed algorithms are discussed in depth.

ReactiveFnJ design has three main components i.e., 1) *Fork*, 2) *Process* and 3) *Join* as shown in Fig. 2. Using the well-established divide and conquer principle, The Fork component divides the main task into smaller subtasks for parallel processing. Once the main task is subdivided, each subtask is processed independently by *Process* component and intermediate results are produced. Responsibility of the Join module is to combine these intermediate results to produce the result.

3.1 The Fork component

The first component of *ReactiveFnJ* i.e., Fork is designed to read a data file of any size and split it into smaller files where size of each split file is always less than or equal to maximum file-size. Maximum file-size is a configurable parameter and can be set according to the serverless execution environment settings.

Challenge in design

In a conventional way, data records of the input file can be read to create small size file splits. Maximum number of records in a split file can be passed as an input parameter. This design for reading and splitting a file will eventually fail in case the input data file is too big. In other words, reading and carving smaller files will continue till the function itself does not surpass the memory usage and execution time-out limits imposed by service providers. This conventional design challenge for serverless is alleviated by a *Relay Composition* pattern *for Fork* as described below.

Relay Composition **for Fork** - In this design, a big file is read, and split files are created by initiating a recursive *Relay Composition* pattern. This pattern can handle an input file of any size, theoretically, without being execution time-out. To make this composition more efficient, data is read and written in byte chunks of almost fixed number of bytes. Let *MaxSplitSize* (*MSS*) indicate the maximum byte chunk that can be read/written by a single function without execution time-out. MSS is an estimated value and can be calculated empirically to know the maximum number of bytes that a *Single FaaS* function can read/write.

The number of split files N can be calculated as follows:

$$N = \left\lceil \left(\frac{Input\ FileSize}{MSS} \right) \right\rceil$$

$$\text{For } 1, 2 \ldots, N \text{ there are } F_1, F_2, \ldots, F_N \text{ split files where } |F_i| \leq MSS, \forall F_i ;\ i \in Z^+$$

The *Relay Composition* starts reading the first byte chunk and writes it to a new file and triggers the next instance asynchronously. New instance starts reading from the byte position in the file where the previous instance had stopped. The last record in a byte chunk may be incomplete due to variable length records and fixed *MSS* value. The composition design takes care of this case by discarding the partial read record and adjusting *StartByteLocation* parameter value as given in Algorithm 1. *StartByteLocation* works as a relay baton and is used to pass the next read position of the input file to the subsequent instance. Hence the *StartByteLocation* is being relayed in every successive recursive instance till the end of the file is reached. This recursive style in *Relay Composition* where *StartByteLocation* being passed in successive calls as shown in Fig. 3. Thus, a file of any size, having variable length records can be forked in serverless infrastructures without being time-out.

Algorithm 1: Read and Split file of any size

Input: InputFilePath, Byte Chunk Size, OutputFilePath

Output: Split Files

```
1. # Reads and splits input file into multiple files of specified size
2. function splitFile(FilePath inputFile, Number numBytesToRead, FilePath outputFile)
3. open inputFile in read mode,  open outputFile in write mode
4. inputFileSize = get number of bytes in inputFile
5. bytesTillNewLine = seek newLine/EOF after numBytesToRead bytes from startPosition
6. while (startPosition + numBytesToRead + bytesTillNewLine < inputFileSize )
7. bytesData = readChunk from startPosition till (startPosition +
8. numBytesToRead + bytesTillNewLine) position
9. write bytesData to outputFile, close outputFile,open new outputFile
10. startPosition = (startPosition + numBytesToRead + bytesTillNewLine)
11. if (startPosition + numBytesToRead + bytesTillNewLine < inputFileSize ) then
12. bytesTillNewLine = seek newLine/EOF after numBytesToRead bytes from startPosition
13. else
14. #remaining file size is less than the bytes to read
15. bytesData = readChunk from startPosition till file end
16. write bytesData to outputFile, close outputFile
17. end if
18. end while
19. end
```

Each recursive instance creates a new split file that can be processed by the Process component in the design pipeline.

3.2 The Process component

The second component of *ReactiveFnJ* is designed to perform computation on all the splits carved in the previous step. All the split files are processed in parallel by independent functions. In essence, the design of *Process* component harnesses the power of scalability offered by a serverless infrastructure. Hence, parallel processing of sub-parts helps in reducing the overall execution time in serverless. This component is the realization of a serverless idea of dynamically creating a compute cluster on demand without any overhead of cluster management. This component can implement use cases where

computation is data independent and therefore, can be executed in parallel. Some exemplary use cases include eCommerce, clickstream analytics, contact centre, legacy app modernization, and DevOps functions. In Algorithm 2, in-memory sort on a split file has been demonstrated but it could be any computation required to run in parallel.

Algorithm 2: In-memory sort of an Individual file
Input: InputFilePath, OutputFilePath
Output: Sorted OutputFile
1. #Sorts the input file in ascending order
2. function sortFile(FilePath inputFile, FilePath outputFile)
3. Open inputFile in read mode
4. For each Row in inputFile do
5. Add row to dataList
6. For End
7. Sort data in dataList in ascending order
8. Open outputFile in write mode
9. For each datarow in dataList do
10. Write datarow to outputFile
11. For End
12. End

3.3 The Join component

The results of the *Process* component are combined in the Join component to converge the result. The main job of this component is to join the processed split files in parallel. As per the design, in the first iteration, all available split files will be paired first and then joined to create a single file. The join component keeps on iterating this till a single file is left as shown in Fig. 2. Design of *Join* component is extremely efficient as number of iterations are growing in a binary logarithmic fashion as shown in Fig. 4 below. The number of iterations *i* required to join N number of files can be determined using our formula $2^{i-1} < N \leq 2^i$ where $i \in Z^+$.

The design of *Join* component was the biggest challenge we faced. Being a serverless component, the *Join* design should adhere to the following serverless principles i) complete decentralized scheduling, ii) reactive inter-module communication, iii) pure asynchronous push-based communication approach. To

claim a component to be truly serverless, it should not use any external orchestration service to synchronize its execution workflows along with the principles mentioned above.

Challenge in design:

The conventional way of joining N processed files in parallel is not a viable design in a serverless environment. In the traditional design, all processed files are joined in parallel in pairs. Each pair of files initiates the event-driven merging process, and a sorted file is created. Newly built joined files are again ready for join and this process will continue till a single file is left as shown in Fig. 5. This typical design works well when the size of a joined file can be handled under the limits of a serverless environment. But it shall eventually fail when file size starts growing as the joining process exceeds the serverless execution time limit. This design challenge is resolved by a reactive *Master-Worker Composition* (MWC) pattern as described below.

Master-Worker Composition **for Join -** MWC is an innovative design solution to handle the exponential increase in file size, the main cause of execution time-out, during the joining process. The main highlight of this design is that it can join any number of files without any constraint on file size.

In MWC, a parallel recursive join process is formulated which is devoid of time-out constraint. In this composition, we have two important modules, first is *MasterReactiveMerge* (MRM) and other one is *WorkerRecursiveMerge* (WRM). The MRM is responsible for initiating a join between a pair of split files. As MRM is reactive so whenever a pair of files is available for join, it is invoked automatically. Event-driven characteristic of serverless architecture helps MRM to run in parallel for each pair of files. Every instance of the MRM module calls a WRM module, responsible for joining two files. First invocation of this module keeps on joining the two files till the size of the joined file reaches a limit called *JoinSafeLimit* which is passed as a parameter to this module. Files joined by WRM could be of any size so its multiple sequential instances may be required to join files of big size. As the file size increases, the number of instances increases proportionally. One instance of WRM joins two files upto *JoinSafeLimit* and before getting time-out calls next WRM instance with appropriate parameters like read positions for both the files. Two files are read and joined to build a single joined file by one or more WRM instances, where the resultant file is appended by each instance. The key design features of MWC are as follows:

1. The Joining process is completely automatic where the number of files and their sizes are not known in advance.
2. There is no central controller responsible for joining files. The design of this component accomplishes the complete decentralized scheduling goal.
3. No additional messaging-service or shared-memory is used for inter-module communication. The design is based on event-driven architecture and makes use of trigger-based communication.
4. The design uses both sequential and parallel composition of serverless functions and synchronizes the burst parallelism.
5. The design ensures that executions are never double billed as design is based on pure asynchronous push-based communication approach. We understand that this design is an example of state-of-the-art

serverless function composition.
6. No external rendezvous server, ad-hoc orchestrator services and current serverless orchestration systems are used in design for task scheduling and synchronization. The design is completely based on the reactive programming model which is highly recommended for serverless.

4. Implementation

Amazon AWS Lambda is a mature FaaS platform, so we opted it for our experimental implementation. The *ReactiveFnJ* model is developed using FaaS and BaaS services provided by AWS.

To validate the design of *ReactiveFnJ*, we implement Split-Sort-Merge (SSM) case study. In this study, the main goal is to sort a data file of any size having variable length records. SSM is a perfect case study based on the generic Fork-Process-Join model of parallel processing. In SSM, Fork component split input file into smaller files, Process component does in-memory sorting of individual split files, and Join component merges the sorted files. We have typically chosen this use case to sort a very big text file that conventionally cannot be sorted using the resources limitations of a single serverless function container. There are many solutions available for handling these scenarios like Cloud IaaS, Hadoop Map Reduce, on-premises cluster etc. But we understand, serverless Function as a Service (FaaS) is the most attractive and methodical option because of its simplicity, billing flexibility and inherent elasticity. Study and implementation of this prototype aims to leverage existing event-based technology of serverless architectures to enable triggered compositions in complex workflows.

4.1 General overview of SSM

In the implementation of SSM, following assorted AWS services are used - i) AWS Simple Storage Service (Amazon S3) for storage of input file, split files, intermediate sorted files and final resultant sorted file, ii) AWS Lambda for fast, containerized, stateless execution of following sub-tasks of workflow: a) splitting b) sorting and c) merging, and iii) Amazon S3 Event Notifications to send event messages to coordinate stateless Lambda functions, iv) AWS Identity and Access Management (IAM) to manage and secure access to various AWS resources [35], v) AWS CloudWatch to monitor and observe data in the form of logs, metrics, and events for complete Fork- Join pipeline.

We implemented the AWS Lambda functions using Python 3.8, which readily offers great library support to manage critical operations like creation, deletion of S3 bucket folders at runtime, setting/retrieving input/output file path, read/write CSV format files etc. A brief description of important python modules used is shown in Table 1.

Table 1: Important Python modules used in implementation of SSM

Python Module Name	Brief Description
csv	CSV (Comma Separated Values) is a simple file format to store tabular data (numbers and text) in plain text. This module implements classes to read and write tabular data in CSV format. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.
os	Provides functions for interacting with the operating system i.e., functions for creating and removing a directory (folder), fetching its contents, changing, and identifying the current directory etc. OS comes under Python's standard utility modules. This module provides a portable way of using operating system-dependent functionality.
botocore	Botocore is a low-level interface to a growing number of Amazon Web Services. The botocore package is the foundation for the AWS CLI as well as boto3.
boto3	It provides Amazon Web Services (AWS) SDK for Python. *boto3* is built atop of library *botocore*. It enables Python developers to create, configure, and manage AWS services, such as S3. The SDK provides an object-oriented API as well as low-level access to AWS services.
s3fs	Pythonic file interface to S3. It builds on top of *botocore*. The top-level class *S3FileSystem* holds connection information and allows typical file-system operations.
json	The *JSON* (JavaScript Object Notation) is a standardized format that allows exchanging of data across the internet. It is a lightweight data interchange format used to work with JSON data.

4.2 Lambda Function Outline

A list of AWS Lambda functions written by us for the SSM implementation are shown in Table 2.

Table 2: Lambda functions used in the Implementation of SSM

S.No	AWS Lambda	Description
1	BL_ReadAndSplit (λBR&S)	Recursive function that reads and split a data file of any size using *Relay Composition* pattern. It uses byte level read/write operations.
2	Sort (λS)	*Sort* function implements in-memory sorting of a split file. It runs in parallel to sort the data contained in the split files.
3	MasterReactiveMerge (λMRM)	Master function responsible for initiating first instance of worker i.e., WorkerRecursiveMerge. It runs in parallel to initiate pair-wise merging of all the sorted split files.
4	WorkerRecursiveMerge (λWRM)	Worker function performs recursive merging of a pair of sorted files of any size. This function also runs in parallel.

The interaction among all Lambda functions of the SSM pipeline is based on AWS S3 triggers and is shown in Table 3 below:

Table 3: AWS S3 Triggers configuration for each Lambda function

TriggerName	Events	Filter	Type	Lambda Function
SplitInput	PUT	*input/*, .csv	Lambda	BL_ReadAndSplit
ProcessInput	PUT	*to_process/*, .csv	Lambda	Sort
JoinInput	PUT	*to_join/*, .csv	Lambda	MasterReactiveMerge
MergeInput	PUT	*to_merge/*, .csv	Lambda	WorkerRecursiveMerge

4.3 Specifics of Lambda functions

In this section, Lambda functions of the SSM pipeline have been comprehensively discussed.

Our implementation uses five AWS S3 folders namely 1) *input* – for storing input file, 2) *to_process* – for storing file splits, 3) *to_join* - for storing sorted split files, intermediate merged files and final merged file, 4) *to_merge* – for temporarily storing pairs of files undergoing merge, and 5) *archive* – to archive input file after successful processing. The complete architecture design of SSM is shown in Figure 6 below:

4.3.1 BL_ReadAndSplit: A large file can be read recursively to overcome the restriction of execution time-out for an AWS Lambda function. The recursive Lambda function, λBR&S, reads a byte chunk and writes it to a new file. It is invoked when an input file is uploaded to the folder "/input" of S3 bucket. Data byte chunks are read from the input file and split files are created in a new folder "to_process" (created at runtime). A configurable parameter *ByteChunkSize* is used by this function that determines the size of split files. Just before getting time-out, this function asynchronously invokes itself by passing the new read byte position in the input file. First instance of λBR&S creates the first split file and invokes the next instance to create the next split file and so on. This process continues till the end of the file. This recursive implementation works perfectly for any size of input file. Invoking a function asynchronously using *lambda.invoke* and setting *Invocation-type* flag as "Event", place invoke requests in Lambda service queue for processing the requests as they arrive. Keeping *lambda.invoke* as the last statement in this function's code eliminates double billing as it would not force the Lambda to wait for each invocation to finish. Thus, recursive invocations of this function where each invocation communicates and coordinates with each other, attains "No Double Billing" condition for the serverless function composition mechanism of ST Trilemma [26].

After the successful completion of this function, the input file is deleted from its folder and is moved to a "\input_archive" folder in S3 bucket. Naming convention of a split file carries two important attributes of information i.e, i) Total number of Splits and ii) File Split Number as shown in Figure 7. Split file name and achieve folder helps in detecting and debugging failures, if any, via AWS CloudWatch log.

4.3.2 Sort: Sort is the AWS Lambda function implemented for sorting a split file. All split files invoke λS on their creation and resultant sorted files are stored in the "/to_join" folder of S3 bucket. Hence λS instances will run in parallel. The coordination between *BL_ReadAndSplit* and *Sort* is done through AWS S3 event notification. In the timeline, one *ReadAndSplit* Lambda function reads a large file and creates multiple small files. Small files will be created in sequence one after another and each of these files will also get sorted and stored in a new S3 bucket folder as shown in Figure 6. Choreography of all these tasks is fully event-driven.

4.3.3 Parallel Reactive Merge is a serverless solution to initiate merging of two sorted files of any size. This implementation solves the challenge of merging files when the size of sorted files starts growing and the execution time-out limit reached before the merging process completes. So Parallel Reactive Merge overcomes the serverless execution time-out constraint. For this, two Lambda functions are implemented: 1) MasterReactiveMerge(λMRM) 2) WorkerRecursiveMerge(λWRM).

In this implementation, a parallel reactive merging process is devised that will complete the merging without being time-out. A Lambda function MasterReactiveMerge is invoked whenever a sorted file is dropped in the *to_join* folder of S3 bucket. This Lambda function picks up another file available in *to_join* for merging, moves these two files to the *to_merge* folder and invokes the first instance of WRM. The WRM merges records from two sorted files and writes them to a new file. The WRM takes input parameters as: i) Working Directory, ii) File1, iii) File2 iv) Start Position in File1, v) Start Position in File2, vi) Output File, and vii) Byte Threshold Value. First invocation of WRM carries Start Position for both the input files as zero. Before getting time-out, WRM calls itself and the next instance starts appending records to the same file created by the first instance. Hence λWRM, recursively calls itself till the single merged file is created as depicted in Figure 8. Last instance of λWRM moves the merged file to the "/to_join" folder and deletes both the input files. One instance of λMRM followed by one or more instances of λWRM is initiated for every pair of files in "/to_join" and this parallel merging process continues till a single file is left in the "/to_join" folder.

5. Evaluation

In this section, we will describe the quantitative assessment for our proposed design model. This will facilitate us to demonstrate the feasibility of our algorithmic approach in designing fully choregraphed Fork-Join workflow in a serverless architecture. This research work also led us to find answers to aforementioned research questions

RQ1: Serverless design using *Relay Composition* pattern can be successfully used for long running computation. Our Fork component allows us to conclude that use cases where processing time might exceed the execution time limit can also make use of serverless technology. Results of our experiments are presented in Table 4 where *MaxSplitSize* is taken as 450 KB.

Table 4
Experimental Results - Single Lambda v/s Relay Composition pattern

Input File Size (KB)	Design Used	Time Out	No. of Files	(No. of Files x *MaxSplitSize* (KB)) + Size of last split file (KB)
200	*SF	No	1	N/A
300	SF	No	1	N/A
600	**SF**	**Yes**	**1**	**N/A**
600	**RC	No	2	(1x450) + 150
700	RC	No	2	(1x450) + 250
800	RC	No	2	(1x450) + 350
1000	RC	No	3	(2x450) + 100
1200	RC	No	3	(2x450) + 300
1400	RC	No	4	(3x450) + 50
1600	RC	No	4	(3x450) + 250

*SF- Single FaaS

**RC- *Relay Composition*

RQ2: Execution a complex, burst-parallel workflow can be addressed programmatically without using any external orchestration service. The algorithmic approach is a priori more powerful considering the availability of basic control flow instructions in any serverless runtime [30]. Our design model can handle burst-parallel workload by utilizing auto scalability of serverless alongwith algorithmic design. Successful implementation of Fork-Join workflow without using any external orchestration service proves this argument.Results of SSM pipeline based on *ReactiveFnJ* are given in Figure 9.

RQ3: It is possible to design a serverless system using pure event-driven architecture for complex workflows. The *ReactiveFnJ*, a pure event-driven system, using two novel serverless design patterns i.e., *Relay Composition* and *Master-Worker Composition*. These patterns are used in prototypical implementation of SSM pipeline that initiates when an input file gets uploaded in the AWS S3 bucket. Subsequent steps like splitting, sorting, and merging are triggered automatically without any human arbitration. This shows that complex parallel systems can be designed in a pure event-driven architecture and hence answers RQ3.

To summarize our results, the *ReactiveFnJ* is a choreographed design model for the development of distributed applications based on Fork-Join workflow with serverless architectures. This design can handle long running tasks by overcoming execution time-out constraint and is not dependent on any external orchestration service for its function composition.

6. Discussion And Lessons

Our model for Fork-Join workflow can be best utilized in serverless technology as this technology provides auto-scalability, built-in fault tolerance, availability, and abstraction of underlying infrastructure. To make Fork-Join workflow available, we build *ReactiveFnJ*, a serverless composition model. The *ReactiveFnJ* uses an innovative design that is purely event-driven, reactive, ST-Safe, choreographed composition model that conquers the hard time-limit forced by serverless environment. This model is completely generic, and several other workflows/use cases can be implemented by substituting specialized FaaS functions in SSM pipeline.

With the intent to develop other complex compute intensive and data parallel applications, patterns analogous to our innovative composition patterns i.e., *Relay Composition* and *Master-Worker Composition* should be discovered. We have articulated following general design guidelines that will help serverless system designers and architects to engineer their applications in an efficient and effective manner.

- Identification of iterative patterns in a long-running task. An iterative pattern can be implemented as a recursive serverless function where each function asynchronously invokes the next instance by relaying the appropriate parameters till the task is completed.

- Recognition of computing patterns that are data independent and hence, can be executed in parallel. These patterns can be implemented as serverless functions runnable in parallel exploiting on-demand scale features of serverless infrastructure.

- For coordination between serverless functions *Master-Worker Composition* is applied. This is a *Master-Worker Composition* where a master function is invoked automatically on an event and is responsible for invoking and dispatching parameters to Worker. The Worker recursively calls itself till the task is completed. In this composition Master is reactive in nature and Worker is recursive in its functioning

7. Related Work

Composition of serverless functions is sparsely covered in the current scientific literature but immensely important in practice [24]. Serverless frameworks must pave the way for complex function composition mechanisms to build responsive, compute intensive, and burst-parallel distributed serverless applications [8].

In some previous studies, researchers have explored and harnessed the power of FaaS in parallel processing applications. For massively parallel computations systems like PyWren [19] and ExCamera [11], have used their own external ad-hoc orchestration services to synchronize the parallel execution of cloud functions. PyWren utilized a polling mechanism for the Amazon S3 bucket to consolidate its

results. ExCamera configured an additional external server to synchronize the interactions of parallel running executions. Dai et al., [10] proposed a trigger-based computing framework called 'Domino'. They exploited both synchronous and asynchronous mechanisms to coordinate triggers. Workflow management systems like Apache Airflow, Oozie [18] and Camunda [32] exist but they depend on a dedicated long-running stateful execution engine to handle the state machine of the orchestration. Different sequential workflows in varied language runtime environments are implemented in [7] on AWS serverless platform and IBM Cloud Function. In our previous work, we proposed a design approach that exploits data parallelism in serverless infrastructure for massively parallel computations. In this design approach, we implemented a running prototype in AWS Lambda for distributed matrix multiplication use case [6].

Serverless popular function orchestration systems are- AWS Step Functions, Azure Durable Functions, IBM Cloud Functions, and Google Cloud Functions. They render help to create services involving compositions of serverless functions. But currently they do not support Parallel Fork-Join workflows natively [4]. As per L´opez et al., [2] serverless applications must follow a trigger-based interaction mechanism. They expressed that the FaaS orchestration system should also be event-triggered. It means, in a complex business workflow, the termination of one FaaS should trigger the next function in the pipeline using asynchronous events. Witte et al., [33] illustrated a cloud specific serverless implementation for seismic imaging application. They use AWS Step Function visual workflow for their implementation and exploit the mathematical properties of imaging optimization problem. Shankar et al., [31] developed a serverless system Numpywren for highly parallel linear algebra algorithms. For their implementation, authors introduced a new domain-specific language "LAmbdaPACK". Zhang et al., [36] proposed a new serverless framework "Kappa", that used a check-pointing mechanism to control function execution time-out. This framework simplifies application development and provides concurrency mechanisms to coordinate parallel computations.

Concurrency and parallel computation of serverless are well suited for requirements like sorting a huge data file. Efficient sorting of large data sets is an extensively discussed area as it is central to large businesses, banks, and institutions. Sorting counts for roughly one-fourth of the total computer cycles [25]. Usually, if a data file is small and can fit into the memory limits of a FaaS container, it can be easily programmed for sorting. But in case a file is huge then sorting by a single FaaS will time-out before its completion. Due to resource limitations of a serverless environment, processing of very large files is not supported directly. Recently, Amazon EFS volume mounted to handle such large files [26]. However, latency and additional cost are some challenges around this way-out.

After investigating previous and recent research works, a clear gap surfaced regarding the non-existence of reactive Fork-Join workflow runnable for serverless infrastructures. To the best of our knowledge, ours is the first work that presents the design and implementation of *ReactiveFnJ* - a pure reactive, choreographed, ST-Safe, algorithmic solution and answer the intrinsic constraint of serverless environments. This makes the best use of the scalability and fault tolerance feature of serverless.

ReactiveFnJ is judicious to be used for large-sized datasets where divide and conquer strategies can be applied.

8. Conclusions And Future Work

Current serverless technology provides its users with fine billing granularity and affordability to run arbitrary functions on-demand. We can reap great benefits of serverless when it comes to the runtime scalability of functions. Subtle barriers of this architecture are resource limit of functions, designing complex applications that require state and composability of long running functions.

This paper is an academic endeavour that designs a choreography model *ReactiveFnJ* for Fork-Join workflow in Serverless architectures. This model is pure event-driven, ST-Safe, not limited to serverless time constraints and makes the best use of scalability and fault tolerance features of serverless. This research proves the viability of a serverless design for a complex burst-parallel workflow where every serverless challenge is resolved at algorithmic level rather than using any external orchestration service, shared memory, or messaging services for its task scheduling and synchronization. This solution is platform independent as a result it is free from vendor lock-in problem also. Our model will render help to serverless architects and developers in fabricating compositions based on Fork-Join workflows. We contemplate that each component of the Fork-Join pipeline can be substituted by a variety of serverless functions to generate diverse workflows. The proof-of-concept and evaluation results authenticate that *ReactiveFnJ* can deliver sufficient performance and suggests its adoption in serverless frameworks for large-scale distributed computing.

Abbreviations

FaaS: Function as a Service; ST: Serverless Trilemma; SSM: Split-Sort-Merge; MSS: MaxSplitSize; MWC: Master-Worker Composition; MRM: MasterReactiveMerge; WRM: WorkerRecursiveMerge.

Declarations

Acknowledgements

Not applicable.

Authors’ contributions

All authors take part in the discussion of the work described in this paper. Urmil Bharti contributed to the model design and validation experiment of this work. She drafted the manuscript and coordinated the review task among authors. The author(s) read and approved the final manuscript.

Authors’ information

Ms. Urmil Bharti has done B.Sc. Computer Science (University of Delhi, India), M.Sc. Computer Science (DAVV Indore, India) and M.Phil. Computer Science (MKU, India). She has over 14 years of teaching experience as Assistant Professor in constituent colleges of University of Delhi. Earlier she worked in IT industry for more than 10 years. Her last designation in industry was Senior Quality Analyst. She is currently doing her research in Cloud and Distributed Computing. Her key research area is cloud computing, serverless technology and software engineering. She has authored several national and international research publications.

Dr. Anita Goel is an Associate Professor in Department of Computer Science, Dyal Singh College, University of Delhi, India. She has received her Ph.D. in Computer Science and Masters in Computer Applications from Jamia Millia Islamia and Department of Computer Science (University of Delhi), respectively. She has a work experience of more than 30 years. She is a visiting faculty to Delhi Technological University and NIIT University. From 2009-10, she was Fellow in Computer Science, at Institute of Life Long Learning (ILLL) in University of Delhi. She has served as member of program committee of International conferences like IEEE BigData Congress 2015 and ICWI 2015. She has guided several students for their doctoral studies and has travelled internationally to present research papers. She has authored books in Computer Science and has several national and international research publications.

Dr SC Gupta is B.Tech (EE) from IIT Delhi and has worked at Computer Group at Tata Institute of Fundamental Research and NCSDCT (now C-DAC Mumbai), Till recently, he worked as Deputy Director General, Scientist-G and Head of Training at National Informatics Centre, New Delhi and was responsible for keeping its 3000 scientists/ engineers uptodate in various technologies. He has extensive experience in design and development of large Complex Software Systems. Currently he is a Visiting Faculty at Dept of Computer Science and Engineering, IIT Delhi. His research interests include Software Engineering, Data Bases and Cloud Computing. He has been teaching Cloud Computing at IIT Delhi, which includes emerging disruptive technologies like SDN and SDS. He has guided many M.Tech & PhD Research students in these technologies and has many publications in Software Engineering and Cloud Technology in National and International Conferences and Journals.

Funding

Not applicable.

Availability of data and materials

Not applicable.

Competing interests

The authors declare that they have no competing interests.

References

1. Arfat Y, Usman S, Mehmood R, Katib I (2020) Big data for smart infrastructure design: Opportunities and challenges. Smart Infrastruct Appl 491–518

2. Arjona A, López PG, Sampé J, Slominski A, Villard L (2021) Triggerflow: Trigger-based orchestration of serverless workflows. Futur Gener Comput Syst 124:215–229. doi: 10.1016/j.future.2021.06.004

3. Baldini I, Cheng P, Fink SJ, Mitchell N, Muthusamy V, Rabbah R, Suter P, Tardieu O (2017) The serverless trilemma: Function composition for serverless computing. Onward! 2017 - Proc 2017 ACM SIGPLAN Int Symp New Ideas, New Paradig Reflections Program Software, co-located with SPLASH 2017 89–103. doi: 10.1145/3133850.3133855

4. Barcelona-Pons D, García-López P, Ruiz Á, Gómez-Gómez A, París G, Sánchez-Artigas M (2019) FaAS orchestration of parallel workloads. In: WOSC 2019 - Proceedings of the 2019 5th International Workshop on Serverless Computing, Part of Middleware 2019. pp 25–30

5. Barcelona-Pons D, Sánchez-Artigas M, París G, Sutra P, García-López P (2019) On the FaaS track: Building stateful distributed applications with serverless architectures. Middlew 2019 - Proc 2019 20th Int Middlew Conf 41–54. doi: 10.1145/3361525.3361535

6. Bharti U, Bajaj D, Goel A, Gupta SC (2021) A Novel Design Approach Exploiting Data Parallelism in Serverless Infrastructure. In: Advances in Computing and Network Communications. Springer, pp 247–260

7. Bharti U, Bajaj D, Goel A, Gupta SC (2021) Sequential Workflow in Production Serverless FaaS Orchestration Platform. In: Proceedings of International Conference on Intelligent Computing, Information and Control Systems. pp 681–693

8. Carver B, Zhang J, Wang A, Anwar A, Wu P, Cheng Y (2020) Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In: SoCC 2020 - Proceedings of the 2020 ACM Symposium on Cloud Computing. Association for Computing Machinery, Inc, pp 1–15

9. Christidis A, Davies R, Moschoyiannis S (2019) Serving machine learning workloads in resource constrained environments: A serverless deployment example. Proc - 2019 IEEE 12th Conf Serv Comput Appl SOCA 2019 55–63. doi: 10.1109/SOCA.2019.00016

10. Dai D, Chen Y, Kimpe D, Ross RB (2021) Trigger-Based Incremental Data Processing with Unified Sync and Async Model. IEEE Trans Cloud Comput 9:372–385. doi: 10.1109/TCC.2018.2830348

11. Fouladi S, Wahby RS, Shacklett B, Balasubramaniam KV, Zeng W, Bhalerao R, Sivaraman A, Porter G, Winstein K (2017) Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. Proc 14th USENIX Symp Networked Syst Des Implementation, NSDI 2017 363–376

12. García-López P, Sánchez-Artigas M, Shillaker S, Pietzuch P, Breitgand D, Vernik G, Sutra P, Tarrant T, Ferrer AJ (2019) ServerMix: Tradeoffs and Challenges of Serverless Data Analytics. pp 1–15

13. Garcia Lopez P, Sanchez-Artigas M, Paris G, Barcelona Pons D, Ruiz Ollobarren A, Arroyo Pinto D (2019) Comparison of FaaS orchestration systems. In: Proceedings - 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018. pp 109–114

14. Giménez-Alventosa V, Moltó G, Caballer M (2019) A framework and a performance assessment for serverless MapReduce on AWS Lambda. Futur Gener Comput Syst 97:259–274. doi: 10.1016/j.future.2019.02.057

15. Hassan HB, Barakat SA, Sarhan QI (2021) Survey on serverless computing. J Cloud Comput 10. doi: 10.1186/s13677-021-00253-7

16. Hellerstein JM, Faleiro J, Gonzalez JE, Schleier-Smith J, Sreekanti V, Tumanov A, Wu C (2019) Serverless computing: One step forward, two steps back. In: CIDR 2019 - 9th Biennial Conference on Innovative Data Systems Research

17. Holubiev V, Ihnatiuk B, Voytyuk I (2018) Next-generation serverless system for contextual search based on rich media content. CEUR Workshop Proc 2300:211–214

18. Islam M, Huang AK, Battisha M, Chiang M, Srinivasan S, Peters C, Neumann A, Abdelnur A (2012) Oozie: towards a scalable workflow management system for hadoop. In: Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies. pp 1–10

19. Jonas E, Pu Q, Venkataraman S, Stoica I, Recht B (2017) Occupy the cloud: Distributed computing for the 99%. SoCC 2017 - Proc 2017 Symp Cloud Comput 445–451. doi: 10.1145/3127479.3128601

20. Klimovic A, Wang Y, Kozyrakis C, Stuedi P, Pfefferle J, Trivedi A (2020) Understanding ephemeral storage for serverless analytics. Proc 2018 USENIX Annu Tech Conf USENIX ATC 2018 789–794

21. Kuhlenkamp J, Werner S, Tai S (2020) The Ifs and Buts of Less is More: A Serverless Computing Reality Check. In: Proceedings - 2020 IEEE International Conference on Cloud Engineering, IC2E 2020. pp 154–161

22. Landset S, Khoshgoftaar TM, Richter AN, Hasanin T (2015) A survey of open source tools for machine learning with big data in the Hadoop ecosystem. J Big Data 2:1–36. doi: 10.1186/s40537-015-0032-1

23. Leite LAF, Oliva GA, Nogueira GM, Gerosa MA, Kon F, Milojicic DS (2013) A systematic literature review of service choreography adaptation. Serv Oriented Comput Appl 7:199–216

24. Leitner P, Wittern E, Spillner J, Hummer W (2019) A mixed-method empirical study of Function-as-a-Service software development in industrial practice. J Syst Softw 149:340–359. doi: 10.1016/j.jss.2018.12.013

25. Leu F-C, Tsai Y-T, Tang CY (2000) An efficient external sorting algorithm. Inf Process Lett 75:159–163

26. Obrutsky S (2016) Cloud Storage : Advantages , Disadvantages and Enterprise Solutions for Business. EIT New Zeal

27. Pu Q, Berkeley UC, Venkataraman S, Stoica I, Berkeley UC, Nsdi I (2019) Shuffling , Fast and Slow : Scalable Analytics on Serverless Infrastructure This paper is included in the Proceedings of the. In: Nsdi

28. Ristov S, Pedratscher S, Fahringer T (2021) AFCL: An Abstract Function Choreography Language for serverless workflow specification. Futur Gener Comput Syst 114:368–382. doi: 10.1016/j.future.2020.08.012

29. Rizk A, Poloczek F, Ciucu F (2015) Computable bounds in fork-join queueing systems. Perform Eval Rev 43:335–346. doi: 10.1145/2796314.2745859

30. Sampé J, Vernik G, Sánchez-Artigas M, García-López P (2018) Serverless data analytics in the IBM cloud. Middlew Ind 2018 - Proc 2018 ACM/IFIP/USENIX Middlew Conf (Industrial Track) 1–8. doi: 10.1145/3284028.3284029

31. Shankar V, Krauth K, Pu Q, Jonas E, Venkataraman S, Stoica I, Recht B, Ragan-Kelley J (2018) Numpywren: Serverless Linear Algebra

32. Wiemuth M, Burgert O (2019) A workflow management system for the OR based on the OMG standards BPMN, CMMN, and DMN. In: Medical Imaging 2019: Image-Guided Procedures, Robotic Interventions, and Modeling. p 1095127

33. Witte PA, Louboutin M, Modzelewski H, Jones C, Selvage J, Herrmann FJ (2020) An event-driven approach to serverless seismic imaging in the cloud. IEEE Trans Parallel Distrib Syst 31:2032–2049

34. Yu T, Liu Q, Du D, Xia Y, Zang B, Lu Z, Yang P, Qin C, Chen H (2020) Characterizing serverless platforms with serverlessbench. In: SoCC 2020 - Proceedings of the 2020 ACM Symposium on Cloud Computing. pp 30–44

35. Zahoor E, Asma Z, Perrin O (2017) A formal approach for the verification of AWS IAM access control policies. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer, Cham, pp 59–74

36. Zhang W, Fang V, Panda A, Shenker S (2020) Kappa: A programming framework for serverless computing. In: Proceedings of the 11th ACM Symposium on Cloud Computing. pp 328–343

Figures

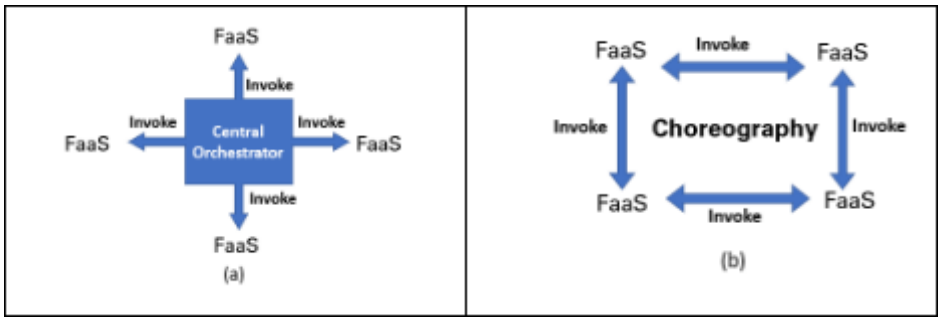

Figure 1

Function Composition Approaches (a) Orchestration (b) Choreography

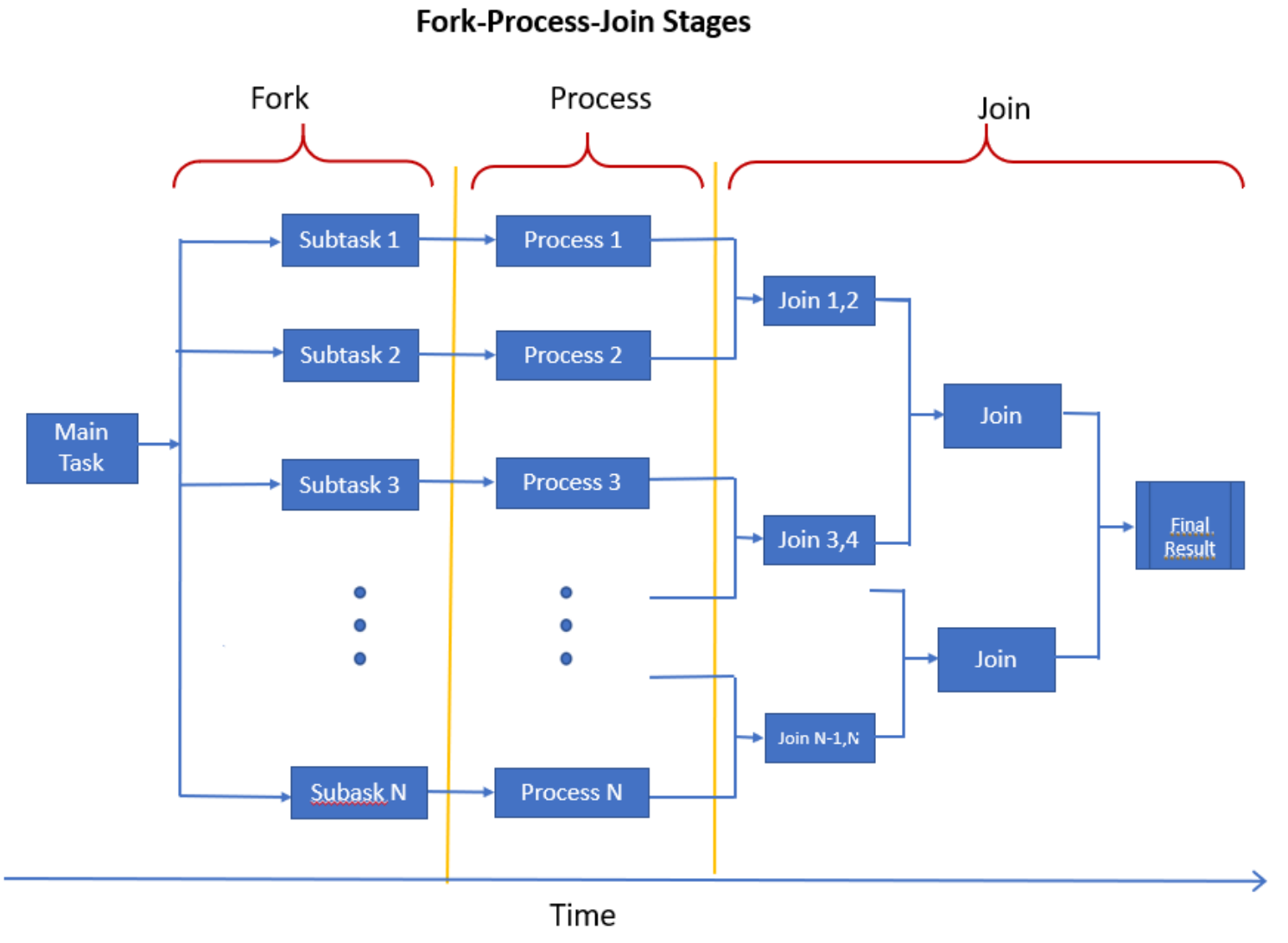

Figure 2

Fork-Process-Join pipeline

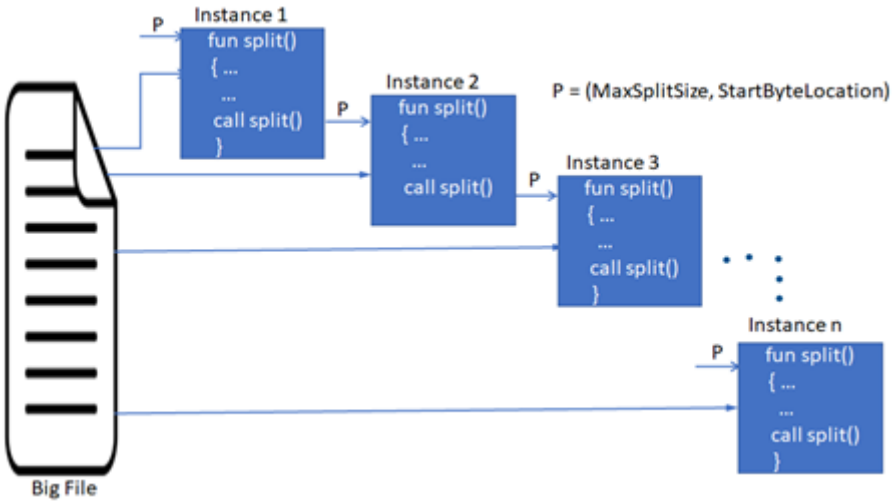

Figure 3

Relay Composition used in spitting a big file

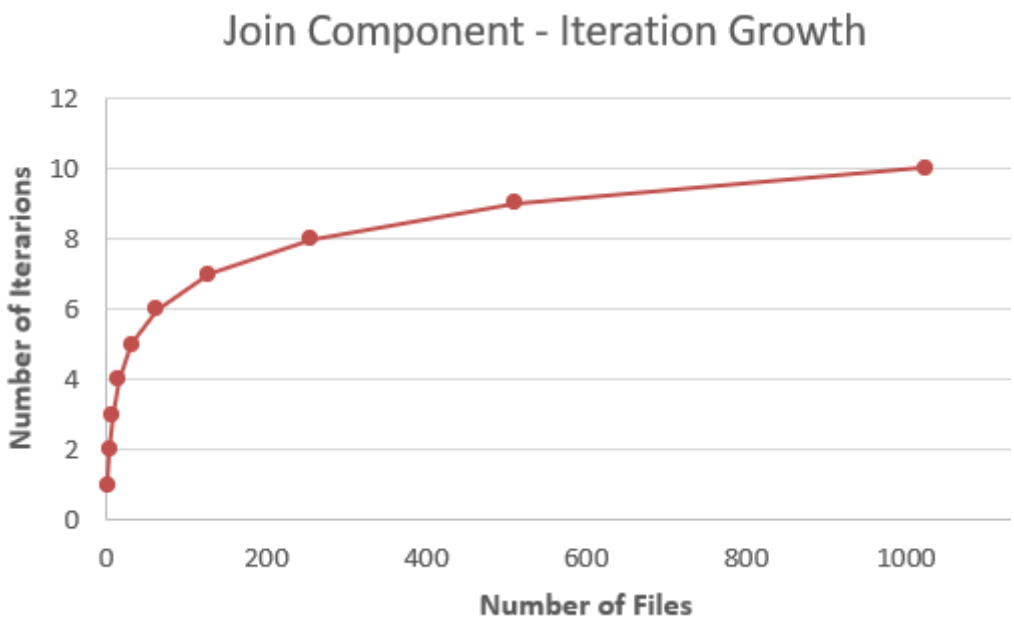

Figure 4

Join Component – Binary Logarithmic Growth Curve

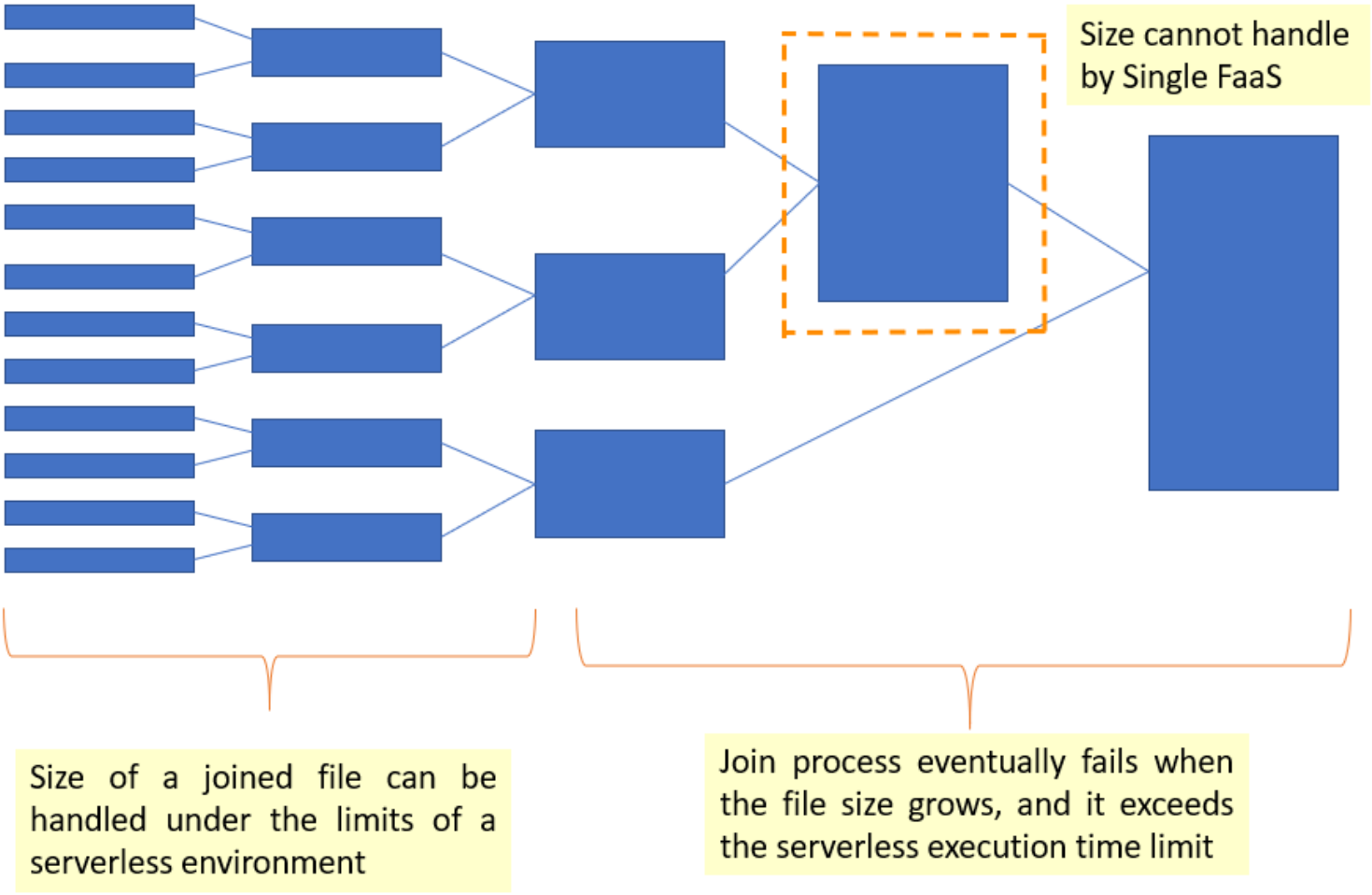

Figure 5

Conventional method to Join N files in Parallel

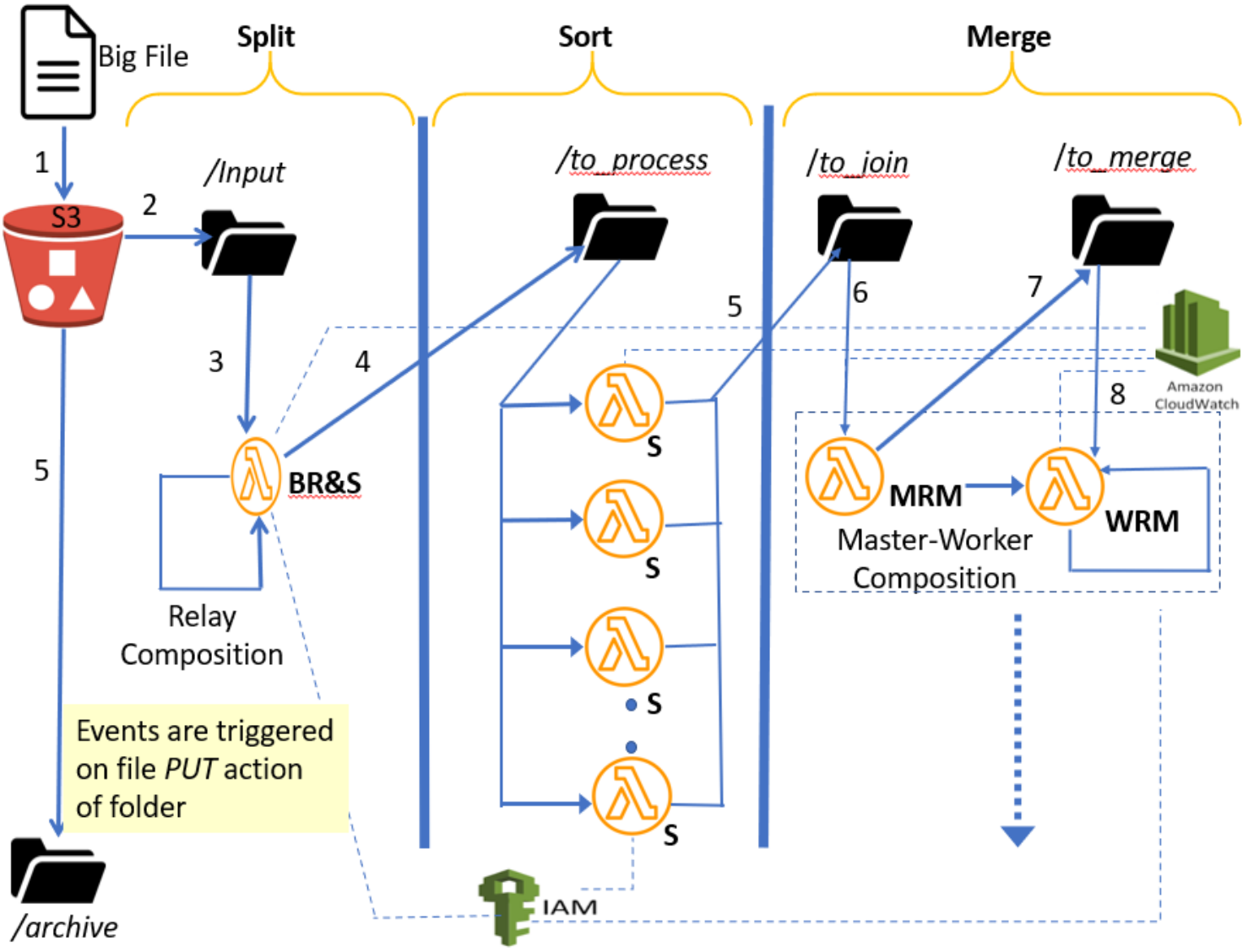

Figure 6

The SSM Architecture

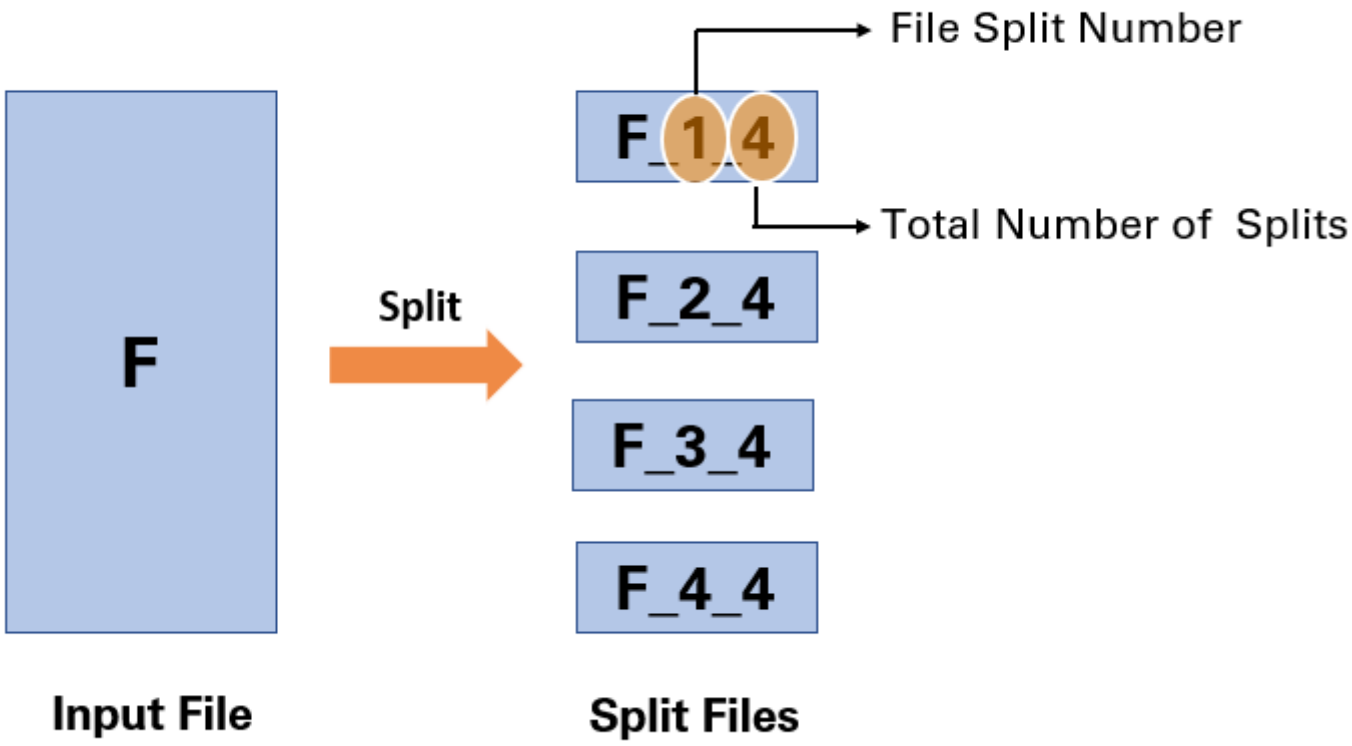

Figure 7

Split File Naming convention

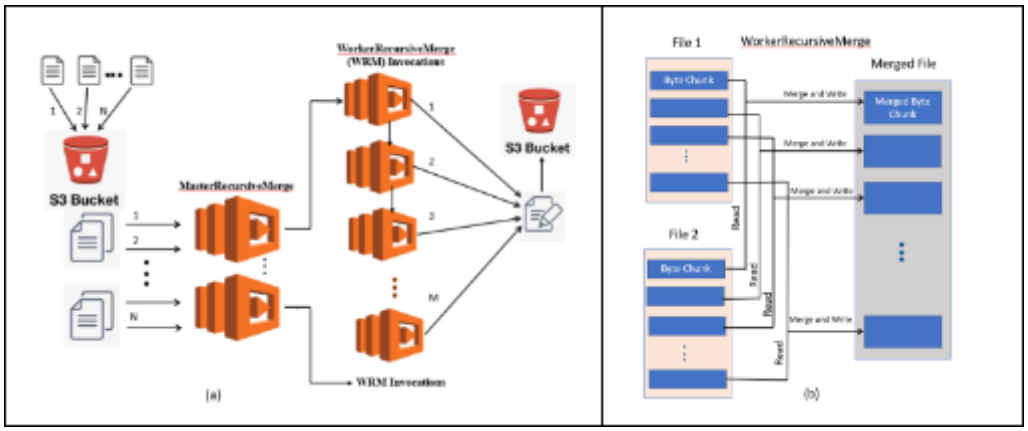

Figure 8

(a) Parallel Reactive Merge (b) File Merge by Worker Recursive Merge

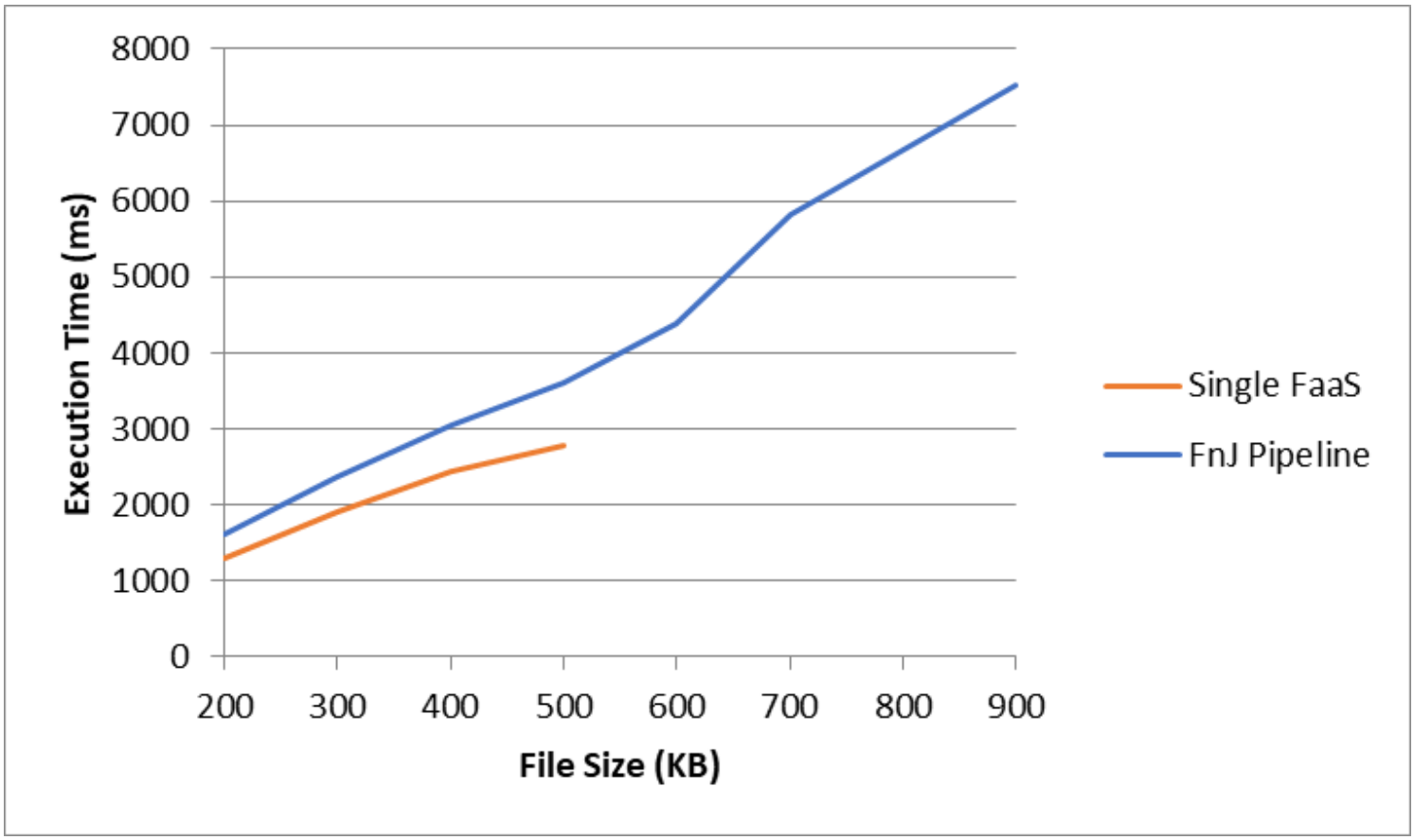

Figure 9

Executions Results of Single Lambda vs FnJ Pipeline